Predicting Customer Churn

Reading the data

The first step is to read the data from the dataset. The dataset consists of various variables such as customer ID, age, gender, account balance, credit score, etc.

In [ ]:
```
import pandas as pd

df = pd.read_csv("Customer_Churn.csv")
df.head(5)
```

Out[ ]:

	CustomerId	Lastname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts
0	15729836	Robinson	646	Spain	Male	32	1	0.00	2
1	15708610	Costa	690	Germany	Male	44	9	100368.63	2
2	15682355	Sabbatini	772	Germany	Male	42	3	75075.31	2
3	15594133	Erskine	697	Spain	Male	62	7	0.00	1
4	15726747	Donaldson	714	France	Male	63	4	138082.16	1

Dropping unnecessary variables

The next step is to drop unnecessary variables from the dataset that are not relevant to detect customer churn. For instance, the customer ID variable does not provide any useful information in detecting customer churn, and hence it can be dropped.

In [ ]:
```
df.drop(['CustomerId', 'Lastname'], axis = 1, inplace = True)
df.head(5)
```

Out[ ]:

	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMe
0	646	Spain	Male	32	1	0.00	2	1	
1	690	Germany	Male	44	9	100368.63	2	0	
2	772	Germany	Male	42	3	75075.31	2	1	
3	697	Spain	Male	62	7	0.00	1	1	
4	714	France	Male	63	4	138082.16	1	0	

Dummy-encode categorical variables

Categorical variables such as gender, etc., need to be encoded to numeric values.

In [ ]:
```
df = pd.get_dummies(df, drop_first = True)
df.head(5)
```

Out[ ]:

	CreditScore	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary
0	646	32	1	0.00	2	1	0	183289.2
1	690	44	9	100368.63	2	0	0	35342.3
2	772	42	3	75075.31	2	1	0	92888.5
3	697	62	7	0.00	1	1	0	129188.1
4	714	63	4	138082.16	1	0	1	166677.5

Scaling the data

The next step is to scale the data in the range of (0,1) using the MinMaxScaler() transformation. Scaling the data helps in avoiding bias towards variables with a higher magnitude.

In [ ]:
```
## standardize
from sklearn.preprocessing import StandardScaler, MinMaxScaler

sc = MinMaxScaler()

## fit
X = sc.fit_transform(df.drop('Churned', axis = 1))
```

In [ ]:
```
X.min()
```

Out[ ]:
```
0.0
```

In [ ]:
```
y = df.Churned
y
```

Out[ ]:
```
0       0
1       0
2       1
3       1
4       0
       ..
8995    0
8996    0
8997    0
8998    1
8999    0
Name: Churned, Length: 9000, dtype: int64
```

Fitting an AutoEncoder architecture

The next step is to fit an AutoEncoder architecture to the scaled data. Remember an AutoEncoder is an unsupervised learning technique that learns the underlying structure of the data by reducing its dimensionality. It consists of an encoder network that maps the input data to a lower-dimensional latent space and a decoder network that reconstructs the original data from the latent space.

In [ ]:
```
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
```

In [ ]:
```
X.shape
```

Out[ ]:
```
(9000, 11)
```

In [ ]:
```
## AUTOENCODER
## Inputlayer :11
## Outputlayer :11, with activation = 'sigmoid' because we used the MinMaxScalar

autoencoder = keras.Sequential([
    layers.Input([11]), ## number of predictors
    layers.Dense(6, activation = 'relu'),
    layers.Dropout(rate = 0.1),
    layers.Dense(3, activation = 'relu'),## bottle neck/latent space
    layers.Dropout(rate = 0.1),
    layers.Dense(6, activation = 'relu'),
    layers.Dropout(rate = 0.1),
    layers.Dense(11, activation = 'sigmoid'),## number of predictors , 0-1 = sigmoid
])
```

In [ ]:
```
## Compile
autoencoder.compile(optimizer = keras.optimizers.Adam(learning_rate = 0.01), loss = 'r
```

In [ ]:
```
## Fit
## For this to be an autoencoder all u need to do is fit on X and X
autoencoder.fit(X,X, epochs = 50, batch_size = 200, validation_split = 0.1)
```

```
Epoch 1/50
41/41 [==============================] - 3s 14ms/step - loss: 0.1288 - val_loss: 0.10
43
Epoch 2/50
41/41 [==============================] - 0s 7ms/step - loss: 0.1044 - val_loss: 0.088
4
Epoch 3/50
41/41 [==============================] - 0s 5ms/step - loss: 0.0963 - val_loss: 0.078
4
Epoch 4/50
41/41 [==============================] - 0s 7ms/step - loss: 0.0890 - val_loss: 0.069
8
Epoch 5/50
41/41 [==============================] - 0s 9ms/step - loss: 0.0842 - val_loss: 0.065
3
Epoch 6/50
41/41 [==============================] - 0s 8ms/step - loss: 0.0811 - val_loss: 0.062
0
Epoch 7/50
41/41 [==============================] - 0s 10ms/step - loss: 0.0785 - val_loss: 0.06
03
Epoch 8/50
41/41 [==============================] - 0s 9ms/step - loss: 0.0766 - val_loss: 0.058
4
Epoch 9/50
41/41 [==============================] - 0s 8ms/step - loss: 0.0742 - val_loss: 0.056
4
Epoch 10/50
41/41 [==============================] - 0s 7ms/step - loss: 0.0733 - val_loss: 0.055
6
Epoch 11/50
41/41 [==============================] - 0s 9ms/step - loss: 0.0723 - val_loss: 0.054
5
Epoch 12/50
41/41 [==============================] - 0s 11ms/step - loss: 0.0720 - val_loss: 0.05
42
Epoch 13/50
41/41 [==============================] - 0s 8ms/step - loss: 0.0717 - val_loss: 0.055
1
Epoch 14/50
41/41 [==============================] - 0s 5ms/step - loss: 0.0708 - val_loss: 0.052
3
Epoch 15/50
41/41 [==============================] - 0s 5ms/step - loss: 0.0706 - val_loss: 0.052
6
Epoch 16/50
41/41 [==============================] - 0s 5ms/step - loss: 0.0700 - val_loss: 0.053
8
Epoch 17/50
41/41 [==============================] - 0s 4ms/step - loss: 0.0699 - val_loss: 0.052
2
Epoch 18/50
41/41 [==============================] - 0s 5ms/step - loss: 0.0691 - val_loss: 0.051
6
Epoch 19/50
41/41 [==============================] - 0s 7ms/step - loss: 0.0694 - val_loss: 0.052
3
Epoch 20/50
41/41 [==============================] - 0s 5ms/step - loss: 0.0687 - val_loss: 0.051
0
```

```
Epoch 21/50
41/41 [==============================] - 0s 5ms/step - loss: 0.0693 - val_loss: 0.052
0
Epoch 22/50
41/41 [==============================] - 0s 5ms/step - loss: 0.0692 - val_loss: 0.050
5
Epoch 23/50
41/41 [==============================] - 0s 6ms/step - loss: 0.0695 - val_loss: 0.052
2
Epoch 24/50
41/41 [==============================] - 0s 8ms/step - loss: 0.0689 - val_loss: 0.052
2
Epoch 25/50
41/41 [==============================] - 0s 6ms/step - loss: 0.0688 - val_loss: 0.052
1
Epoch 26/50
41/41 [==============================] - 0s 5ms/step - loss: 0.0681 - val_loss: 0.053
2
Epoch 27/50
41/41 [==============================] - 0s 8ms/step - loss: 0.0685 - val_loss: 0.052
3
Epoch 28/50
41/41 [==============================] - 0s 6ms/step - loss: 0.0680 - val_loss: 0.053
1
Epoch 29/50
41/41 [==============================] - 0s 4ms/step - loss: 0.0683 - val_loss: 0.053
6
Epoch 30/50
41/41 [==============================] - 0s 5ms/step - loss: 0.0674 - val_loss: 0.051
8
Epoch 31/50
41/41 [==============================] - 0s 6ms/step - loss: 0.0680 - val_loss: 0.052
7
Epoch 32/50
41/41 [==============================] - 0s 5ms/step - loss: 0.0679 - val_loss: 0.054
8
Epoch 33/50
41/41 [==============================] - 0s 5ms/step - loss: 0.0675 - val_loss: 0.054
0
Epoch 34/50
41/41 [==============================] - 0s 4ms/step - loss: 0.0677 - val_loss: 0.050
7
Epoch 35/50
41/41 [==============================] - 0s 5ms/step - loss: 0.0676 - val_loss: 0.050
6
Epoch 36/50
41/41 [==============================] - 0s 3ms/step - loss: 0.0676 - val_loss: 0.050
7
Epoch 37/50
41/41 [==============================] - 0s 3ms/step - loss: 0.0684 - val_loss: 0.048
8
Epoch 38/50
41/41 [==============================] - 0s 3ms/step - loss: 0.0673 - val_loss: 0.054
9
Epoch 39/50
41/41 [==============================] - 0s 3ms/step - loss: 0.0670 - val_loss: 0.056
0
Epoch 40/50
41/41 [==============================] - 0s 3ms/step - loss: 0.0675 - val_loss: 0.055
2
```

```
Epoch 41/50
41/41 [==============================] - 0s 3ms/step - loss: 0.0676 - val_loss: 0.056
3
Epoch 42/50
41/41 [==============================] - 0s 3ms/step - loss: 0.0676 - val_loss: 0.054
8
Epoch 43/50
41/41 [==============================] - 0s 3ms/step - loss: 0.0669 - val_loss: 0.052
1
Epoch 44/50
41/41 [==============================] - 0s 3ms/step - loss: 0.0676 - val_loss: 0.057
5
Epoch 45/50
41/41 [==============================] - 0s 3ms/step - loss: 0.0673 - val_loss: 0.054
9
Epoch 46/50
41/41 [==============================] - 0s 3ms/step - loss: 0.0675 - val_loss: 0.055
9
Epoch 47/50
41/41 [==============================] - 0s 3ms/step - loss: 0.0676 - val_loss: 0.054
0
Epoch 48/50
41/41 [==============================] - 0s 3ms/step - loss: 0.0672 - val_loss: 0.057
8
Epoch 49/50
41/41 [==============================] - 0s 3ms/step - loss: 0.0667 - val_loss: 0.060
0
Epoch 50/50
41/41 [==============================] - 0s 3ms/step - loss: 0.0671 - val_loss: 0.050
4
```

Out[ ]:

```
<keras.callbacks.History at 0x7ff8e520d990>
```

Determining the 500 most likely to churn customers

After fitting the AutoEncoder, we can use it to predict the customers who are most likely to churn. We can do this by selecting the customers whose reconstruction error is high. The reconstruction error is the difference between the original input data and the reconstructed data obtained from the AutoEncoder. We can sort the customers based on their reconstruction error and select the 500 customers with the highest error as the most likely to churn.

In [ ]:

```
## The Reconstruction error
## the predictions are the reconstruction of the variables
predictions = autoencoder.predict(X)
```

```
282/282 [==============================] - 0s 1ms/step
```

In [ ]:

```
## how things were reconstructed
pd.DataFrame(sc.inverse_transform(predictions))
```

Out[ ]:

	0	1	2	3	4	5	6	7
0	644.085571	37.799175	4.763194	73140.742188	1.526535	9.998091e-01	0.251400	105703.445312
1	651.760010	38.335217	4.972935	81248.335938	1.538330	7.446102e-01	0.322325	100570.234375
2	651.762634	38.335194	4.972907	81247.140625	1.538346	7.446117e-01	0.322788	100571.835938
3	644.124268	37.798306	4.762522	73115.679688	1.526788	9.998106e-01	0.258022	105734.875000
4	652.349365	38.276878	4.866066	55170.718750	1.634177	1.406282e-32	0.997044	104788.875000
...	...	...	...	...	...	...	...	...
8995	644.231567	37.809395	4.767321	73258.757812	1.526786	9.997784e-01	0.252601	105607.625000
8996	661.089661	38.262138	4.875755	77190.625000	1.596354	7.495748e-01	0.998606	106132.828125
8997	659.510437	38.139111	4.829956	74427.531250	1.595359	9.345653e-01	0.998640	107416.062500
8998	651.747559	38.335274	4.973098	81246.960938	1.538267	7.445813e-01	0.320076	100562.015625
8999	649.233826	38.314945	4.968323	74975.125000	1.542635	4.137923e-06	0.233137	100535.351562

9000 rows × 11 columns

MSE Reconstruction Error

In [ ]:

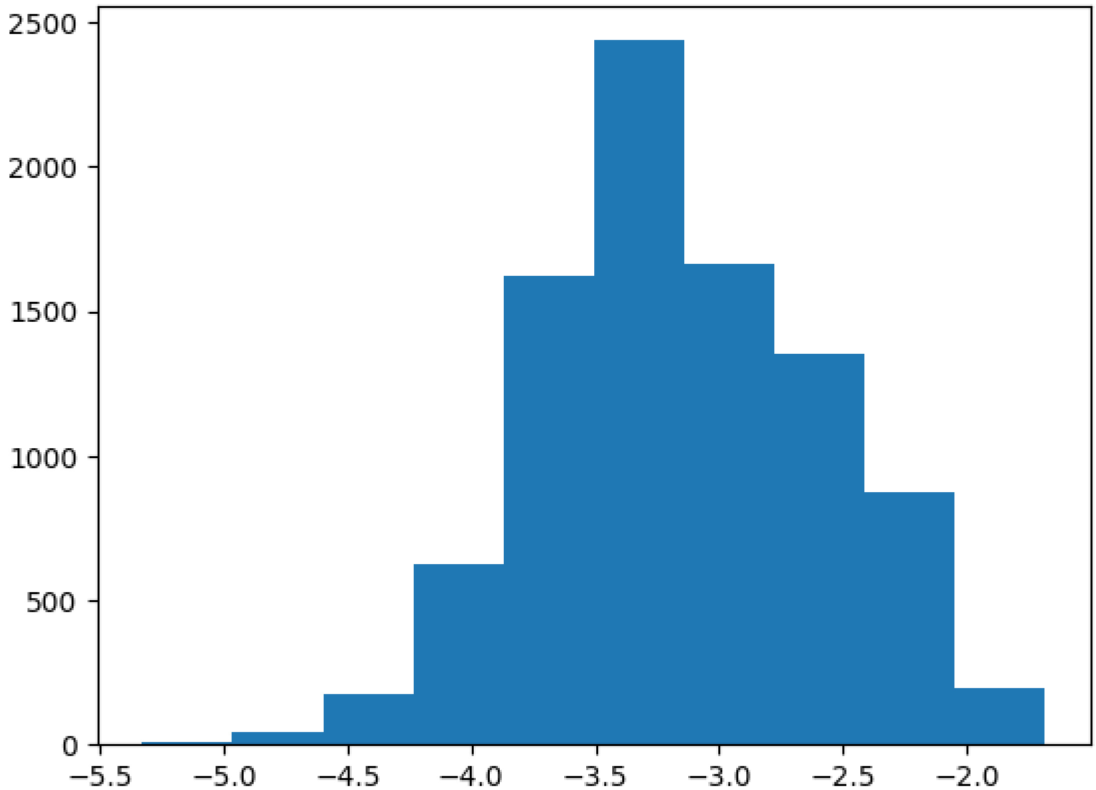
```
import numpy as np
mse = np.mean((X - predictions)**2, axis = 1)
mse
## these are the reconstructed amount
```

Out[ ]:

```
array([0.04287146, 0.13489593, 0.07249676, ..., 0.01971494, 0.12849466,
       0.02695782])
```

In [ ]:

```
plt.hist(np.log(mse));
```



In [ ]:
```
df_error = df
df_error['Reconst_Error'] = mse
```

In [ ]:
```
# We sort them according to the error so that largest reconstruction sre on top
df_error = df_error.sort_values('Reconst_Error', ascending= False)
df_error.head(5)
```

Out[ ]:

	CreditScore	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSa
7980	606	65	10	126306.64	3	0	0	786
4454	645	68	9	0.00	4	1	1	17635
639	797	55	10	0.00	4	1	1	4941
8144	720	57	1	162082.31	4	0	0	2714
8213	350	39	0	109733.20	2	0	0	12360

500 customers with the highest error as the most likely to churn.

In [ ]:
```
top500 = df_error.head(500)
```

In [ ]:
```
top500.Churned.sum()
```

Out[ ]:
```
185
```

Verifying the Predictions

Once we have selected the 500 most likely to churn customers, we need to determine how many of them did actually churn. We can compare our predictions with the actual churn status of the customers to calculate the accuracy of our model.

In [ ]:
```
## Number of customers total that churned in dataset
count_ones = df['Churned'].sum()
count_ones
```

Out[ ]:
```
1822
```

In [ ]:
```
185/1822
```

Out[ ]:
```
0.10153677277716795
```

The 500 observations show us that there were about 10% customers that were detected to be churned.

In [ ]:
```
## Convert the notebook to HTML
!jupyter nbconvert --to html /content/Project_Part1.ipynb
```

```
[NbConvertApp] Converting notebook /content/Project_Part1.ipynb to html
[NbConvertApp] Writing 655892 bytes to /content/Project_Part1.html
```